Efficient CCG Parsing: A* versus Adaptive Supertagging

Efficient CCG Parsing: A* versus Adaptive Supertagging

Michael Auli
School of Informatics
University of Edinburgh
m.auli@sms.ed.ac.uk

Adam Lopez
HLTCOE
Johns Hopkins University
alopez@cs.jhu.edu

Abstract

We present a systematic comparison and combination of two orthogonal techniques for efficient parsing of Combinatory Categorial Grammar (CCG). First we consider adaptive supertagging, a widely used approximate search technique that prunes most lexical categories from the parser's search space using a separate sequence model. Next we consider several variants on A*, a classic exact search technique which to our knowledge has not been applied to more expressive grammar formalisms like CCG. In addition to standard hardware-independent measures of parser effort we also present what we believe is the first evaluation of A* parsing on the more realistic but more stringent metric of CPU time. By itself, A* substantially reduces parser effort as measured by the number of edges considered during parsing, but we show that for CCG this does not always correspond to improvements in CPU time over a CKY baseline. Combining A* with adaptive supertagging decreases CPU time by 15% for our best model.

1 Introduction

Efficient parsing of Combinatorial Categorial Grammar (CCG; Steedman, 2000) is a longstanding problem in computational linguistics. Even with practical CCG that are strongly context-free (Fowler and Penn, 2010), parsing can be much harder than with Penn Treebank-style context-free grammars, since the number of nonterminal categories is generally much larger, leading to increased grammar constants. Where a typical Penn Treebank grammar may have fewer than 100 nonterminals (Hockenmaier and Steedman, 2002), we found that a CCG grammar derived from CCGbank contained nearly 1600. The same grammar assigns an average of 26 lexical categories per word, resulting in a very large space of possible derivations.

The most successful strategy to date for efficient parsing of CCG is to first prune the set of lexical categories considered for each word, using the output of a *supertagger*, a sequence model over these categories (Bangalore and Joshi, 1999; Clark, 2002). Variations on this approach drive the widely-used, broad coverage C&C parser (Clark and Curran, 2004; Clark and Curran, 2007). However, pruning means approximate search: if a lexical category used by the highest probability derivation is pruned, the parser will not find that derivation (§2). Since the supertagger enforces no grammaticality constraints it may even prefer a sequence of lexical categories that cannot be combined into *any* derivation (Figure 1). Empirically, we show that supertagging improves efficiency by an order of magnitude, but the tradeoff is a significant loss in accuracy (§3).

Can we improve on this tradeoff? The line of investigation we pursue in this paper is to consider more efficient *exact* algorithms. In particular, we test different variants of the classical A* algorithm (Hart et al., 1968), which has met with success in Penn Treebank parsing with context-free grammars (Klein and Manning, 2003; Pauls and Klein, 2009a; Pauls and Klein, 2009b). We can substitute A* for standard CKY on either the unpruned set of lexical categories, or the pruned set resulting from su-

Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics, pages 1577–1585,
Portland, Oregon, June 19-24, 2011.

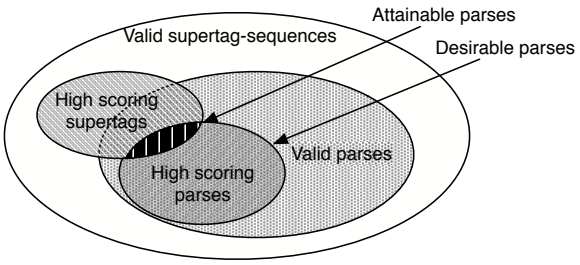
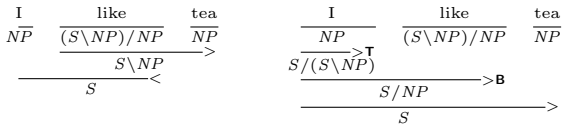


Figure 1: The relationship between supertagger and parser search spaces based on the intersection of their corresponding tag sequences.

pertagging. Our empirical results show that on the unpruned set of lexical categories, heuristics employed for context-free grammars show substantial speedups in hardware-independent metrics of parser effort (§4). To understand how this compares to the CKY baseline, we conduct a carefully controlled set of timing experiments. Although our results show that improvements on hardware-independent metrics do not always translate into improvements in CPU time due to increased processing costs that are hidden by these metrics, they also show that when the lexical categories are pruned using the output of a supertagger, then we can still improve efficiency by 15% with A* techniques (§5).

2 CCG and Parsing Algorithms

CCG is a lexicalized grammar formalism encoding for each word lexical categories which are either basic (eg. NN, JJ) or complex. Complex lexical categories specify the number and directionality of arguments. For example, one lexical category (of over 100 in our model) for the transitive verb *like* is $(S\backslash NP_2)/NP_1$, specifying the first argument as an NP to the right and the second as an NP to the left. In parsing, adjacent spans are combined using a small number of binary combinatory rules like forward application or composition on the spanning categories (Steedman, 2000; Fowler and Penn, 2010). In the first derivation below, $(S\backslash NP)/NP$ and NP combine to form the spanning category $S\backslash NP$, which only requires an NP to its left to form a complete sentence-spanning S. The second derivation uses type-raising to change the category type of *I*.



Because of the number of lexical categories and their complexity, a key difficulty in parsing CCG is that the number of analyses for each span of the sentence quickly becomes extremely large, even with efficient dynamic programming.

2.1 Adaptive Supertagging

Supertagging (Bangalore and Joshi, 1999) treats the assignment of lexical categories (or *supertags*) as a sequence tagging problem. Once the supertagger has been run, lexical categories that apply to each word in the input sentence are pruned to contain only those with high posterior probability (or other figure of merit) under the supertagging model (Clark and Curran, 2004). The posterior probabilities are then discarded; it is the extensive pruning of lexical categories that leads to substantially faster parsing times.

Pruning the categories in advance this way has a specific failure mode: sometimes it is not possible to produce a sentence-spanning derivation from the tag sequences preferred by the supertagger, since it does not enforce grammaticality. A workaround for this problem is the *adaptive supertagging* (AST) approach of Clark and Curran (2004). It is based on a step function over supertagger beam ratios, relaxing the pruning threshold for lexical categories whenever the parser fails to find an analysis. The process either succeeds and returns a parse after some iteration or gives up after a predefined number of iterations. As Clark and Curran (2004) show, most sentences can be parsed with a very small number of supertags per word. However, the technique is inherently approximate: it will return a lower probability parse under the parsing model if a higher probability parse can only be constructed from a supertag sequence returned by a subsequent iteration. In this way it prioritizes speed over accuracy, although the tradeoff can be modified by adjusting the beam step function.

2.2 A* Parsing

Irrespective of whether lexical categories are pruned in advance using the output of a supertagger, the CCG parsers we are aware of all use some vari-

ant of the CKY algorithm. Although CKY is easy to implement, it is *exhaustive*: it explores all possible analyses of all possible spans, irrespective of whether such analyses are likely to be part of the highest probability derivation. Hence it seems natural to consider exact algorithms that are more efficient than CKY.

A* search is an agenda-based best-first graph search algorithm which finds the lowest cost parse exactly without necessarily traversing the entire search space (Klein and Manning, 2003). In contrast to CKY, items are not processed in topological order using a simple control loop. Instead, they are processed from a priority queue, which orders them by the product of their inside probability and a heuristic estimate of their outside probability. Provided that the heuristic never underestimates the true outside probability (i.e. it is *admissible*) the solution is guaranteed to be exact. Heuristics are model specific and we consider several variants in our experiments based on the CFG heuristics developed by Klein and Manning (2003) and Pauls and Klein (2009a).

3 Adaptive Supertagging Experiments

Parser. For our experiments we used the generative CCG parser of Hockenmaier and Steedman (2002). Generative parsers have the property that all edge weights are non-negative, which is required for A* techniques.[1] Although not quite as accurate as the discriminative parser of Clark and Curran (2007) in our preliminary experiments, this parser is still quite competitive. It is written in Java and implements the CKY algorithm with a global pruning threshold of 10^{-4} for the models we use. We focus on two parsing models: `PCFG`, the baseline of Hockenmaier and Steedman (2002) which treats the grammar as a PCFG (Table 1); and `HWDep`, a headword dependency model which is the best performing model of the parser. The PCFG model simply generates a tree top down and uses very simple structural probabilities while the `HWDep` model conditions node expansions on headwords and their lexical categories.

Supertagger. For supertagging we used Dennis Mehay's implementation, which follows Clark (2002).[2] Due to differences in smoothing of the supertagging and parsing models, we occasionally drop supertags returned by the supertagger because they do not appear in the parsing model [3].

Evaluation. All experiments were conducted on CCGBank (Hockenmaier and Steedman, 2007), a right-most normal-form CCG version of the Penn Treebank. Models were trained on sections 2-21, tuned on section 00, and tested on section 23. Parsing accuracy is measured using labelled and unlabelled predicate argument structure recovery (Clark and Hockenmaier, 2002); we evaluate on *all* sentences and thus penalise lower coverage. All timing experiments reported in the paper were run on a 2.5 GHz Xeon machine with 32 GB memory and are averaged over ten runs[4].

3.1 Results

Supertagging has been shown to improve the speed of a generative parser, although little analysis has been reported beyond the speedups (Clark, 2002) We ran experiments to understand the time/accuracy tradeoff of adaptive supertagging, and to serve as baselines.

Adaptive supertagging is parametrized by a beam size β and a dictionary cutoff k that bounds the number of lexical categories considered for each word (Clark and Curran, 2007). Table 3 shows both the standard beam levels (`AST`) used for the C&C parser and looser beam levels: `AST-covA`, a simple extension of `AST` with increased coverage and `AST-covB`, also increasing coverage but with better performance for the `HWDep` model.

Parsing results for the AST settings (Tables 4 and 5) confirm that it improves speed by an order of magnitude over a baseline parser without AST. Perhaps surprisingly, the number of parse failures decreases with AST in some cases. This is because the parser prunes more aggressively as the search space increases.[5]

[1] Indeed, all of the past work on A* parsing that we are aware of uses generative parsers (Pauls and Klein, 2009b, *inter alia*).

[2] http://code.google.com/p/statopenccg

[3] Less than 2% of supertags are affected by this.

[4] The timing results reported differ from an earlier draft since we used a different machine

[5] Hockenmaier and Steedman (2002) saw a similar effect.

Expansion probability	$p(exp\|P)$	exp $\in$ {leaf, unary, left-head, right-head}
Head probability	$p(H\|P, exp)$	H is the head daughter
Non-head probability	$p(S\|P, exp, H)$	S is the non-head daughter
Lexical probability	$p(w\|P)$	

Table 1: Factorisation of the PCFG model. H,P, and S are categories, and w is a word.

Expansion probability	$p(exp\|P, c_P\#w_P)$	exp $\in$ {leaf, unary, left-head, right-head}
Head probability	$p(H\|P, exp, c_P\#w_P)$	H is the head daughter
Non-head probability	$p(S\|P, exp, H\#c_P\#w_P)$	S is the non-head daughter
Lexcat probability	$p(c_S\|S\#P, H, S)$	$p(c_{TOP}\|P{=}TOP)$
Headword probability	$p(w_S\|c_S\#P, H, S, w_P)$	$p(w_{TOP}\|c_{TOP})$

Table 2: Headword dependency model factorisation, backoff levels are denoted by '#' between conditioning variables: A # B # C indicates that $\hat{P}(\ldots|A, B, C)$ is interpolated with $\hat{P}(\ldots|A, B)$, which is an interpolation of $\hat{P}\ldots|A, B)$ and $\hat{P}(\ldots|A)$. Variables c_P and w_P represent, respectively, the head lexical category and headword of category P.

Condition	Parameter	Iteration 1	2	3	4	5	6
AST	β (beam width)	0.075	0.03	0.01	0.005	0.001	
	k (dictionary cutoff)	20	20	20	20	150	
AST-covA	β	0.075	0.03	0.01	0.005	0.001	0.0001
	k	20	20	20	20	150	150
AST-covB	β	0.03	0.01	0.005	0.001	0.0001	0.0001
	k	20	20	20	20	20	150

Table 3: Beam step function used for standard (AST) and high-coverage (AST-covA and AST-covB) supertagging.

	Time(sec)	Sent/sec	Cats/word	Fail	UP	UR	UF	LP	LR	LF
PCFG	290	6.6	26.2	5	86.4	86.5	86.5	77.2	77.3	77.3
PCFG (AST)	**65**	**29.5**	1.5	14	**87.4**	85.9	86.6	**79.5**	78.0	78.8
PCFG (AST-covA)	67	28.6	1.5	6	87.3	**86.9**	**87.1**	79.1	**78.8**	**78.9**
PCFG (AST-covB)	69	27.7	1.7	5	87.3	86.2	86.7	79.1	78.1	78.6
HWDep	1512	1.3	26.2	5	**90.2**	**90.1**	**90.2**	**83.2**	**83.0**	**83.1**
HWDep (AST)	**133**	**14.4**	1.5	16	89.8	88.0	88.9	82.6	80.9	81.8
HWDep (AST-covA)	139	13.7	1.5	9	89.8	88.3	89.0	82.6	81.1	81.9
HWDep (AST-covB)	155	12.3	1.7	7	90.1	88.7	89.4	83.0	81.8	82.4

Table 4: Results on CCGbank section 00 when applying adaptive supertagging (AST) to two models of a generative CCG parser. Performance is measured in terms of parse failures, labelled and unlabelled precision (LP/UP), recall (LR/UR) and F-score (LF/UF). Evaluation is based only on sentences for which each parser returned an analysis.

3.2 Efficiency versus Accuracy

The most interesting result is the effect of the speedup on accuracy. As shown in Table 6, the vast majority of sentences are actually parsed with a very tight supertagger beam, raising the question of whether many higher-scoring parses are pruned.[6] Despite this, labeled F-score improves by up to 1.6 F-measure for the PCFG model, although it harms accuracy for HWDep as expected.

In order to understand this effect, we filtered section 00 to include only sentences of between 18 and 26 words (resulting in 610 sentences) for which

[6]Similar results are reported by Clark and Curran (2007).

	Time(sec)	Sent/sec	Cats/word	Fail	UP	UR	UF	LP	LR	LF
PCFG	326	7.4	25.7	29	85.9	85.4	85.7	76.6	76.2	76.4
PCFG (AST)	**82**	**29.4**	1.5	34	**86.7**	84.8	85.7	**78.6**	76.9	77.7
PCFG (AST-covA)	85	28.3	1.6	15	86.6	85.5	86.0	78.5	**77.5**	**78.0**
PCFG (AST-covB)	86	27.9	1.7	14	86.6	**85.6**	**86.1**	78.1	77.3	77.7
HWDep	1754	1.4	25.7	30	**90.2**	**89.3**	**89.8**	**83.5**	**82.7**	**83.1**
HWDep (AST)	**167**	**14.4**	1.5	27	89.5	87.6	88.5	82.3	80.6	81.5
HWDep (AST-covA)	177	13.6	1.6	14	89.4	88.1	88.8	82.2	81.1	81.7
HWDep (AST-covB)	188	12.8	1.7	14	89.7	88.5	89.1	82.5	81.4	82.0

Table 5: Results on CCGbank section 23 when applying adaptive supertagging (AST) to two models of a CCG parser.

β	Cats/word	Parses	%
0.075	1.33	1676	87.6
0.03	1.56	114	6.0
0.01	1.97	60	3.1
0.005	2.36	15	0.8
$0.001_{k=150}$	3.84	32	1.7
Fail		16	0.9

Table 6: Breakdown of the number of sentences parsed for the HWDep (AST) model (see Table 4) at each of the supertagger beam levels from the most to the least restrictive setting.

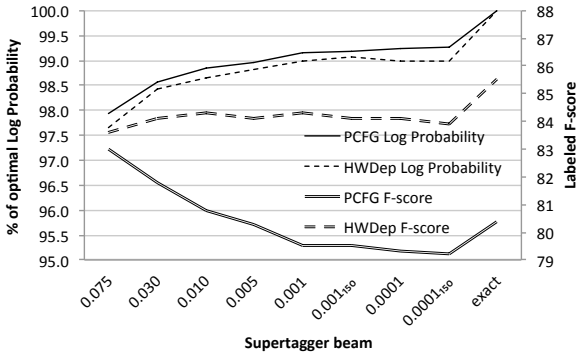


Figure 2: Log-probability of parses relative to exact solution vs. labelled F-score at each supertagging beam-level.

we can perform exhaustive search without pruning[7], and for which we could parse without failure at all of the tested beam settings. We then measured the log probability of the highest probability parse found under a variety of beam settings, relative to the log probability of the unpruned exact parse, along with the labeled F-Score of the Viterbi parse under these settings (Figure 2). The results show that PCFG actually finds worse results as it considers more of the search space. In other words, the supertagger can actually "fix" a bad parsing model by restricting it to a small portion of the search space. With the more accurate HWDep model, this does not appear to be a problem and there is a clear opportunity for improvement by considering the larger search space. The next question is whether we can exploit this larger search space without paying as high a cost in efficiency.

[7]The fact that only a subset of short sentences could be exhaustively parsed demonstrates the need for efficient search algorithms.

4 A* Parsing Experiments

To compare approaches, we extended our baseline parser to support A* search. Following (Klein and Manning, 2003) we restrict our experiments to sentences on which we can perform exact search via using the same subset of section 00 as in §3.2. Before considering CPU time, we first evaluate the amount of work done by the parser using three hardware-independent metrics. We measure the number of *edges pushed* (Pauls and Klein, 2009a) and *edges popped*, corresponding to the insert/decrease-key operations and remove operation of the priority queue, respectively. Finally, we measure the number of *traversals*, which counts the number of edge weights computed, regardless of whether the weight is discarded due to the prior existence of a better weight. This latter metric seems to be the most accurate account of the work done by the parser.

Due to differences in the PCFG and HWDep models, we considered different A* variants: for the PCFG model we use a simple A* with a precom-

puted heuristic, while for the the more complex HWDep model, we used a hierarchical A* algorithm (Pauls and Klein, 2009a; Felzenszwalb and McAllester, 2007) based on a simple grammar projection that we designed.

4.1 Hardware-Independent Results: PCFG

For the PCFG model, we compared three agenda-based parsers: EXH prioritizes edges by their span length, thereby simulating the exhaustive CKY algorithm; NULL prioritizes edges by their inside probability; and SX is an A* parser that prioritizes edges by their inside probability times an admissible outside probability estimate.[8] We use the SX estimate devised by Klein and Manning (2003) for CFG parsing, where they found it offered very good performance for relatively little computation. It gives a bound on the outside probability of a nonterminal P with i words to the right and j words to the left, and can be computed from a grammar using a simple dynamic program.

The parsers are tested with and without adaptive supertagging where the former can be seen as performing exact search (via A*) over the pruned search space created by AST.

Table 7 shows that A* with the SX heuristic decreases the number of edges pushed by up to 39% on the unpruned search space. Although encouraging, this is not as impressive as the 95% speedup obtained by Klein and Manning (2003) with this heuristic on their CFG. On the other hand, the NULL heuristic works better for CCG than for CFG, with speedups of 29% and 11%, respectively. These results carry over to the AST setting which shows that A* can improve search even on the highly pruned search graph. Note that A* only saves work in the final iteration of AST, since for earlier iterations it must process the entire agenda to determine that there is no spanning analysis.

Since there are many more categories in the CCG grammar we might have expected the SX heuristic to work better than for a CFG. Why doesn't it? We can measure how well a heuristic bounds the true cost in terms of *slack*: the difference between the true and estimated outside cost. Lower slack means that the heuristic bounds the true cost better and guides us to the exact solution more quickly. Figure 3 plots the average slack for the SX heuristic against the number of words in the outside context. Comparing this with an analysis of the same heuristic when applied to a CFG by Klein and Manning (2003), we find that it is less effective in our setting[9]. There is a steep increase in slack for outside contexts with size more than one. The main reason for this is because a single word in the outside context is in many cases the full stop at the end of the sentence, which is very predictable. However for longer spans the flexibility of CCG to analyze spans in many different ways means that the outside estimate for a nonterminal can be based on many high probability outside derivations which do not bound the true probability very well.

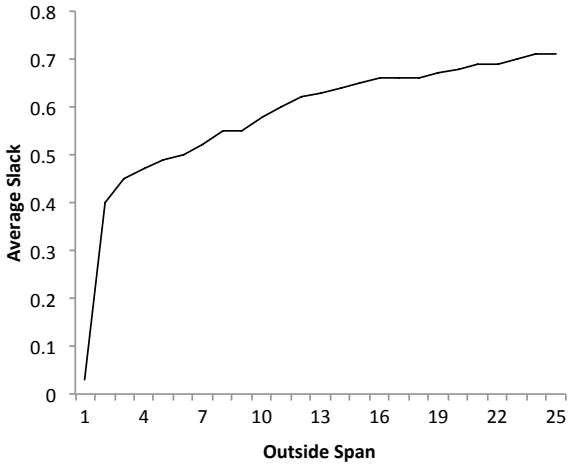


Figure 3: Average slack of the SX heuristic. The figure aggregates the ratio of the difference between the estimated outside cost and true outside costs relative to the true cost across the development set.

4.2 Hardware-Independent Results: HWDep

Lexicalization in the HWDep model makes the precomputed SX estimate impractical, so for this model we designed two hierarchical A* (HA*) variants based on simple grammar projections of the model. The basic idea of HA* is to compute Viterbi inside probabilities using the easier-to-parse projected

[8] The NULL parser is a special case of A*, also called uniform cost search, which in the case of parsing corresponds to Knuth's algorithm (Knuth, 1977; Klein and Manning, 2001), the extension of Dijkstra's algorithm to hypergraphs.

[9] Specifically, we refer to Figure 9 of their paper which uses a slightly different representation of estimate sharpness

Parser	Edges pushed				Edges popped				Traversals			
	Std	%	AST	%	Std	%	AST	%	Std	%	AST	%
EXH	34	100	6.3	100	15.7	100	4.2	100	133.4	100	13.3	100
NULL	24.3	71	4.9	78	13.5	86	3.5	83	113.8	85	11.1	83
SX	**20.9**	**61**	**4.3**	**68**	**10.0**	**64**	**2.6**	**62**	**96.5**	**72**	**9.7**	**73**

Table 7: Exhaustive search (EXH), A* with no heuristic (NULL) and with the SX heuristic in terms of millions of edges pushed, popped and traversals computed using the PCFG grammar with and without adaptive supertagging.

grammar, use these to compute Viterbi outside probabilities for the simple grammar, and then use these as outside estimates for the true grammar; all computations are prioritized in a single agenda following the algorithm of Felzenszwalb and McAllester (2007) and Pauls and Klein (2009a). We designed two simple grammar projections, each simplifying the HWDep model: PCFGProj completely removes lexicalization and projects the grammar to a PCFG, while as LexcatProj removes only the headwords but retains the lexical categories.

Figure 4 compares exhaustive search, A* with no heuristic (NULL), and HA*. For HA*, parsing effort is broken down into the different edge types computed at each stage: We distinguish between the work carried out to compute the inside and outside edges of the projection, where the latter represent the heuristic estimates, and finally, the work to compute the edges of the target grammar. We find that A* NULL saves about 44% of edges pushed which makes it slightly more effective than for the PCFG model. However, the effort to compute the grammar projections outweighs their benefit. We suspect that this is due to the large difference between the target grammar and the projection: The PCFG projection is a simple grammar and so we improve the probability of a traversal less often than in the target grammar.

The Lexcat projection performs worst, for two reasons. First, the projection requires about as much work to compute as the target grammar without a heuristic (NULL). Second, the projection itself does not save a large amount of work as can be seen in the statistics for the target grammar.

5 CPU Timing Experiments

Hardware-independent metrics are useful for understanding agenda-based algorithms, but what we actually care about is CPU time. We were not aware of any past work that measures A* parsers in terms of CPU time, but as this is the real objective we feel that experiments of this type are important. This is especially true in real implementations because the savings in edges processed by an agenda parser come at a cost: operations on the priority queue data structure can add significant runtime.

Timing experiments of this type are very implementation-dependent, so we took care to implement the algorithms as cleanly as possible and to reuse as much of the existing parser code as we could. An important implementation decision for agenda-based algorithms is the data structure used to implement the priority queue. Preliminary experiments showed that a Fibonacci heap implementation outperformed several alternatives: Brodal queues (Brodal, 1996), binary heaps, binomial heaps, and pairing heaps.[10]

We carried out timing experiments on the best A* parsers for each model (SX and NULL for PCFG and HWDep, respectively), comparing them with our CKY implementation and the agenda-based CKY simulation EXH; we used the same data as in §3.2. Table 8 presents the cumulative running times with and without adaptive supertagging average over ten runs, while Table 9 reports F-scores.

The results (Table 8) are striking. Although the timing results of the agenda-based parsers track the hardware-independent metrics, they start at a significant disadvantage to exhaustive CKY with a simple control loop. This is most evident when looking at the timing results for EXH, which in the case of the full PCFG model requires more than twice the time than the CKY algorithm that it simulates. A*

[10] We used the Fibonacci heap implementation at http://www.jgrapht.org

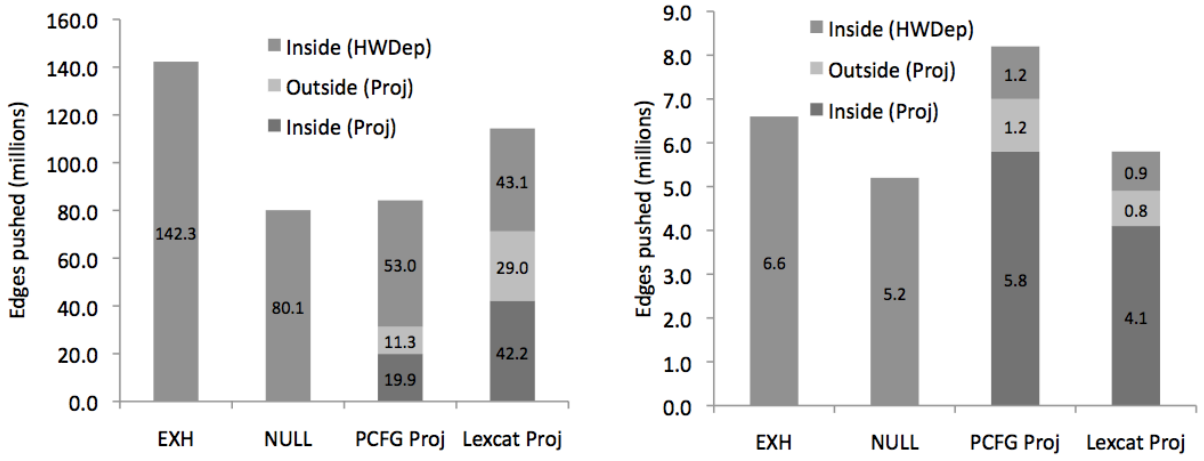


Figure 4: Comparsion between a CKY simulation (EXH), A* with no heuristic (NULL), hierarchical A* (HA*) using two grammar projections for standard search (left) and AST (right). The breakdown of the inside/outside edges for the grammar projection as well as the target grammar shows that the projections, serving as the heuristic estimates for the target grammar, are costly to compute.

	Standard		AST	
	PCFG	HWDep	PCFG	HWDep
CKY	536	24489	34	143
EXH	1251	26889	41	155
A* NULL	1032	21830	36	121
A* SX	889	-	34	-

Table 8: Parsing time in seconds of CKY and agenda-based parsers with and without adaptive supertagging.

	Standard		AST	
	PCFG	HWDep	PCFG	HWDep
CKY	80.4	85.5	81.7	83.8
EXH	79.4	85.5	80.3	83.8
A* NULL	79.6	85.5	80.7	83.8
A* SX	79.4	-	80.4	-

Table 9: Labelled F-score of exact CKY and agenda-based parsers with/without adaptive supertagging. All parses have the same probabilities, thus variances are due to implementation-dependent differences in tiebreaking.

makes modest CPU-time improvements in parsing the full space of the HWDep model. Although this decreases the time required to obtain the highest accuracy, it is still a substantial tradeoff in speed compared with AST.

On the other hand, the AST tradeoff improves significantly: by combining AST with A* we observe a decrease in running time of 15% for the A* NULL parser of the HWDep model over CKY. As the CKY baseline with AST is very strong, this result shows that A* holds real promise for CCG parsing.

6 Conclusion and Future Work

Adaptive supertagging is a strong technique for efficient CCG parsing. Our analysis confirms tremendous speedups, and shows that for weak models, it can even result in improved accuracy. However, for better models, the efficiency gains of adaptive supertagging come at the cost of accuracy. One way to look at this is that the supertagger has good precision with respect to the parser's search space, but low recall. For instance, we might combine both parsing and supertagging models in a principled way to exploit these observations, eg. by making the supertagger output a soft constraint on the parser rather than a hard constraint. Principled, efficient search algorithms will be crucial to such an approach.

To our knowledge, we are the first to measure A* parsing speed both in terms of running time and commonly used hardware-independent metrics. It is clear from our results that the gains from A* do not come as easily for CCG as for CFG, and that agenda-based algorithms like A* must make very large reductions in the number of edges processed to result in realtime savings, due to the added expense of keeping a priority queue. However, we

have shown that A* can yield real improvements even over the highly optimized technique of adaptive supertagging: in this pruned search space, a 44% reduction in the number of edges pushed results in a 15% speedup in CPU time. Furthermore, just as A* can be combined with adaptive supertagging, it should also combine easily with other search-space pruning methods, such as those of Djordjevic et al. (2007), Kummerfeld et al. (2010), Zhang et al. (2010) and Roark and Hollingshead (2009). In future work we plan to examine better A* heuristics for CCG, and to look at principled approaches to combine the strengths of A*, adaptive supertagging, and other techniques to the best advantage.

Acknowledgements

We would like to thank Prachya Boonkwan, Juri Ganitkevitch, Philipp Koehn, Tom Kwiatkowski, Matt Post, Mark Steedman, Emily Thomforde, and Luke Zettlemoyer for helpful discussion related to this work and comments on previous drafts; Julia Hockenmaier for furnishing us with her parser; and the anonymous reviewers for helpful commentary. We also acknowledge funding from EPSRC grant EP/P504171/1 (Auli); the EuroMatrixPlus project funded by the European Commission, 7th Framework Programme (Lopez); and the resources provided by the Edinburgh Compute and Data Facility (`http://www.ecdf.ed.ac.uk/`). The ECDF is partially supported by the eDIKT initiative (`http://www.edikt.org.uk/`).

References

S. Bangalore and A. K. Joshi. 1999. Supertagging: An Approach to Almost Parsing. *Computational Linguistics*, 25(2):238–265, June.

G. S. Brodal. 1996. Worst-case efficient priority queues. In *Proc. of SODA*, pages 52–58.

S. Clark and J. R. Curran. 2004. The importance of supertagging for wide-coverage CCG parsing. In *Proc. of COLING*.

S. Clark and J. R. Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552.

S. Clark and J. Hockenmaier. 2002. Evaluating a wide-coverage CCG parser. In *Proc. of LREC Beyond Parseval Workshop*, pages 60–66.

S. Clark. 2002. Supertagging for Combinatory Categorial Grammar. In *Proc. of TAG+6*, pages 19–24.

B. Djordjevic, J. R. Curran, and S. Clark. 2007. Improving the efficiency of a wide-coverage CCG parser. In *Proc. of IWPT*.

P. F. Felzenszwalb and D. McAllester. 2007. The Generalized A* Architecture. In *Journal of Artificial Intelligence Research*, volume 29, pages 153–190.

T. A. D. Fowler and G. Penn. 2010. Accurate context-free parsing with combinatory categorial grammar. In *Proc. of ACL*.

P. Hart, N. Nilsson, and B. Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *Transactions on Systems Science and Cybernetics*, 4, Jul.

J. Hockenmaier and M. Steedman. 2002. Generative models for statistical parsing with Combinatory Categorial Grammar. In *Proc. of ACL*, pages 335–342. Association for Computational Linguistics.

J. Hockenmaier and M. Steedman. 2007. CCGbank: A corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396.

D. Klein and C. D. Manning. 2001. Parsing and hypergraphs. In *Proc. of IWPT*.

D. Klein and C. D. Manning. 2003. A* parsing: Fast exact Viterbi parse selection. In *Proc. of HLT-NAACL*, pages 119–126, May.

D. E. Knuth. 1977. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6:1–5.

J. K. Kummerfeld, J. Roesner, T. Dawborn, J. Haggerty, J. R. Curran, and S. Clark. 2010. Faster parsing by supertagger adaptation. In *Proc. of ACL*.

A. Pauls and D. Klein. 2009a. Hierarchical search for parsing. In *Proc. of HLT-NAACL*, pages 557–565, June.

A. Pauls and D. Klein. 2009b. k-best A* Parsing. In *Proc. of ACL-IJCNLP*, ACL-IJCNLP '09, pages 958–966.

B. Roark and K. Hollingshead. 2009. Linear complexity context-free parsing pipelines via chart constraints. In *Proc. of HLT-NAACL*.

M. Steedman. 2000. *The syntactic process*. MIT Press, Cambridge, MA.

Y. Zhang, B.-G. Ahn, S. Clark, C. V. Wyk, J. R. Curran, and L. Rimell. 2010. Chart pruning for fast lexicalised-grammar parsing. In *Proc. of COLING*.